Rigel: Flexible Multi-Rate Image Processing Hardware

James Hegarty Ross Daly Zachary DeVito Jonathan Ragan-Kelley Mark Horowitz Pat Hanrahan

Stanford University

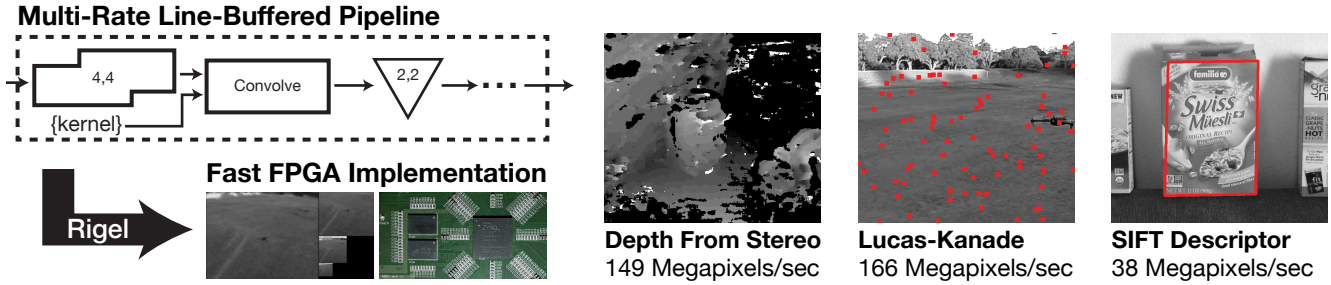


Figure 1: *Rigel takes a flexible multi-rate architecture based on synchronous dataflow and compiles it to efficient FPGA implementations. Our architecture supports pyramid image processing, sparse computations, and space-time implementation tradeoffs. We show depth from stereo, Lucas-Kanade, the SIFT descriptor, and a Gaussian pyramid running at between 20-436 megapixels/second on two FPGA platforms.*

Abstract

Image processing algorithms implemented using custom hardware or FPGAs of can be orders-of-magnitude more energy efficient and performant than software. Unfortunately, converting an algorithm by hand to a hardware description language suitable for compilation on these platforms is frequently too time consuming to be practical. Recent work on hardware synthesis of high-level image processing languages demonstrated that a single-rate pipeline of stencil kernels can be synthesized into hardware with provably minimal buffering. Unfortunately, few advanced image processing or vision algorithms fit into this highly-restricted programming model.

In this paper, we present Rigel[1], which takes pipelines specified in our new multi-rate architecture and lowers them to FPGA implementations. Our flexible multi-rate architecture supports pyramid image processing, sparse computations, and space-time implementation tradeoffs. We demonstrate depth from stereo, Lucas-Kanade, the SIFT descriptor, and a Gaussian pyramid running on two FPGA boards. Our system can synthesize hardware for FPGAs with up to 436 Megapixels/second throughput, and up to 297× faster runtime than a tablet-class ARM CPU.

Keywords: Image processing, domain-specific languages, hardware synthesis, FPGAs, video processing.

Concepts: •**Computing methodologies** → **Image processing;** •**Hardware** → **Hardware description languages and compilation;**

[1] http://rigel-fpga.org

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

SIGGRAPH '16 Technical Paper,, July 24 - 28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4279-7/16/07
DOI: http://dx.doi.org/10.1145/2897824.2925892

1 Introduction

The smartphone revolution has yielded a proliferation of small, cheap, and low power cameras and processors. We think this presents an opportunity for computer vision technology to become pervasive, leading to innovation in new interfaces, augmented reality, and the internet of things.

The challenge is that image processing, particularly computer vision, does a tremendous number of operations per second. For example, a simple brute force depth from stereo algorithm does around 10,000 mathematical operations per pixel [Scharstein and Szeliski 2002]. Running this algorithm on 1080p/60fps video requires 1.2 Terraops/second. While CPU/GPU platforms exist that can perform computation at this rate, they also use a large amount of power, and are not suitable for use in mobile devices. Simply offloading the computation to a datacenter does not solve the problem, since wireless transmission uses 1,000,000× more energy than a local arithmatic operation [Huang et al. 2012].

Often the only way to implement these compute-heavy image processing applications within the power limits of a mobile platform is with custom fixed-function hardware. Prior work has shown that fixed-function hardware implementations of image processing can increase energy efficiency by 500× compared to general-purpose processors [Hameed et al. 2010]. Much of this efficiency comes from reducing the number of indirections through DRAM and SRAM by using custom datapaths and local on-chip storage. DRAM reads use 1,000× the energy of performing an arithmatic operation, so these savings can be significant.

While most developers will not be fabricating custom hardware, FPGAs are a widely-available alternative that have energy efficiency significantly higher than CPUs/GPUs. FPGA boards are available in many different I/O and performance configurations, which makes them a compelling platform for experimentation and prototyping. Many existing cameras such as the Edgertronic high framerate camera, Red cinema camera, and Digital Bolex already use FPGAs to implement part of their image processing pipelines.

Unfortunately, implementing custom hardware and FPGA designs remains inaccessible to most application developers. Conventional

hardware description languages like Verilog require the user to work at an extremely low level, manually dealing with tasks like pipelining and conforming to hardware interfaces, which is tedious and error-prone.

Prior work on the Darkroom system shows how to synthesize a high-level image processing language based on pipelines of stencil computations into custom hardware and FPGA designs [Hegarty et al. 2014]. Darkroom keeps all intermediates on-chip using a provably minimal amount of buffering, which is crucial for energy efficiency. Unfortunately, Darkroom only supports pipelines that operate on one pixel per cycle, which limits it to single-scale image processing algorithms, and couples chip area to algorithm complexity.

In this paper, we present a new multi-rate architecture for image processing based on foundational work in Darkroom and Synchronous Dataflow. We show how a set of simple multi-rate primitives can be used to simultaneously support three features that are crucial to synthesizing hardware for advanced image processing and vision algorithms: pyramid image processing, sparse computations, and space-time implementation tradeoffs.

Pyramid image processing enables efficient implementation of algorithms that operate at different scales. Pyramidal implementations can improve the quality of the result for little extra computational cost [Bouguet 2001; Adelson et al. 1984]. *Sparse computations* improve efficiency by terminating work early that will not improve quality, e.g., in sparse feature matching [Lowe 1999]. Finally, *space-time implementation tradeoffs* allow the user to choose to use less chip area at the expense of more computation time. Supporting space-time tradeoffs is necessary to efficiently implement multi-rate algorithms like pyramids and sparse computations.

This paper makes the following contributions:

- We present the *multi-rate line-buffered pipeline architecture*, based on extensions to Synchronous Dataflow and Darkroom's line-buffered pipeline architecture.
- We show how our multi-rate architecture can be used to implement pyramid image processing, sparse computations, and support space-time implementation tradeoffs.
- We implement depth from stereo, Lucas-Kanade, the SIFT descriptor, and Gaussian pyramids in our architecture, and use our compiler to map these algorithms to two FPGA boards. Our synthesized designs for FPGA have throughputs between 20 megapixels/second and 436 megapixels/seconds. We demonstrate that our system can efficiently support pyramids, sparse computations, and space-time tradeoffs. Finally, we demonstrate a camera test rig we have built to verify our FPGA implementations.

2 Background

2.1 Line-buffered Pipelines

Prior work on fixed-function image processing hardware formalized the *line-buffered pipeline* architecture [Hegarty et al. 2014; Brunhaver 2015]. In a line-buffered pipeline, each kernel can only read from its inputs in a statically-known bounded region around the current pixel called a *stencil*. These stencil reads can be realized in hardware with a small local RAM called a *line buffer* (fig. 2). Restricting kernels to only read stencils allows the compiler to provably minimize the size of the line buffers by changing when kernels are computed relative to each other.

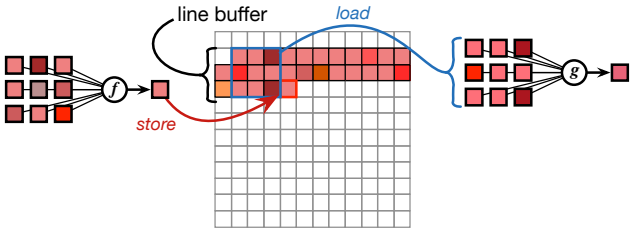


Figure 2: *The line-buffered pipeline architecture requires that each kernel in the pipeline only reads from its inputs in a bounded region around the output pixel.*

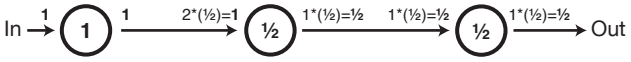
2.2 Dataflow Languages

Dataflow languages model programs as directed graphs of nodes representing computations. Edges in the graph indicate producer-consumer relationships. Each time a node in the graph *fires*, or executes, it consumes a certain number of *tokens* from its inputs, and sends a certain number of tokens to its outputs.

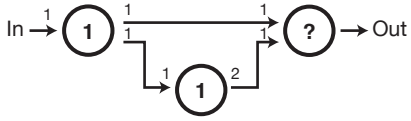
Foundational work in dataflow languages is Synchronous Dataflow (SDF) [Lee and Messerschmitt 1987]. In SDF, each node consumes exactly N input tokens, and produces exactly M output tokens per firing. Here we show these input and output rates (tokens/firing) as annotations on each incoming and outgoing edge:



In order for buffering along each edge to be bounded, it must be the case that the rate of data flowing into and out of each edge is the same. SDF solves for *firing rates* to satisfy this requirement. Firing rates set how frequently nodes fire relative to each other. Effective rates along each edge are then each node's input or output rate multiplied by the node's firing rate. Here we show one possible SDF solution for the previous example. Firing rates are shown inside each circle, and effective rates along edges are calculated:

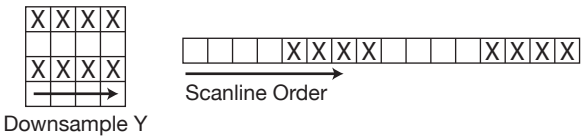


For some SDF graphs, it is impossible to find firing rates that make rates along all edges match. These graphs will always deadlock, because no matter how many tokens are put into the pipeline, there will be a node that is still waiting for data. This can occur, for example, by merging an upsampled stream with the original stream. No matter what firing rate is used for the rightmost node, it will never match all its input rates:



Solving for firing rates of a graph only involves finding the nullspace of a matrix, so it can be computed quickly [Lee and Messerschmitt 1987]. Followup work shows how to implement a SDF computation as statically-scheduled CPU code with provably minimal buffering [Murthy et al. 1997].

Statically-scheduled SDF requires each node to consume and produce exactly its number of input and output tokens each time it fires. While this works well in the 1D domain, many modules in a 2D scanline architecture do not fit into this restriction. For example, a vertical downsample produces pixels in line-sized bursts when executed pixel at a time in scanline order:



Modules like downsample have a known total number of input/output tokens, but a latency (number of firings before each output actually appears) that varies within a bounded number of firings. We call modules with this property *variable-latency* modules.

Embedding variable-latency modules in our architecture requires extensions beyond statically-scheduled SDF. Followup work on improving the functionality of SDF has taken either a static or dynamic scheduling approach. Static scheduling work, such as cyclostatic dataflow, increases the flexibility of SDF but keeps some restrictions that allow for static analysis [Bilsen et al. 1995]. These models keep some or all of SDF's deadlock and buffering properties, at the expense of added scheduling complexity [Bilsen et al. 1995; Murthy and Lee 2002]. Dynamic scheduling approaches such as GRAMPS place no restrictions on the number of tokens that can be produced or consumed each firing, but also cannot prove any properties about deadlock or buffering [Sugerman et al. 2009]. Prior work exists on compiling SDF graphs to hardware (such as [Horstmannshoff et al. 1997]), but to our knowledge no system exists that supports variable-latency modules.

Rigel takes a hybrid approach between SDF and dynamic scheduling, which we call *variable-latency SDF*. We restrict our pipeline to be a Directed Acyclic Graph (DAG) of SDF nodes. However, we allow nodes to have variable latency, and implement the SDF execution in hardware using dynamic scheduling. We use first-in-first-out (FIFO) queues to hide latency variation, creating a graph of kernels that behaves at the top level similarly to a traditional SDF system. This allows us to use SDF to prove that the pipeline will not deadlock, but also support the variable-latency modules we need for our target applications.

2.3 Image Processing Languages

Halide is a CPU/GPU image processing language with a separate algorithm language and scheduling language [Ragan-Kelley et al. 2012]. Halide's scheduling language is used to map the algorithm language into executable code, based on a number of loop transforms. Halide's algorithm and scheduling languages are general, so making scheduling decisions requires either programmer insight, autotuning, or heuristics [Mullapudi et al. 2016]. However, experimenting with different Halide schedules is faster than rewriting the code by hand in lower-level languages like C.

Rigel was inspired by Halide's choice to focus on programmer productivity instead of automated scheduling, which often necessitates a loss in flexibility. Rigel attempts to make an equivalent system for hardware, where we allow the user manual control of a set of flexible and powerful hardware tradeoffs with more convenience and ease of experimentation than is possible in existing hardware languages like Verilog.

2.4 High-Level Synthesis

An emerging technology in recent years is high-level synthesis (HLS), which takes languages such as C or CUDA and compiles them to hardware. For example, Vivado synthesizes a subset of C into a Xilinx FPGA design guided by a number of pragma annotations [Vivado 2016]. In our experience, CPU-targeted image processing code requires extensive modification to perform well with HLS tools.

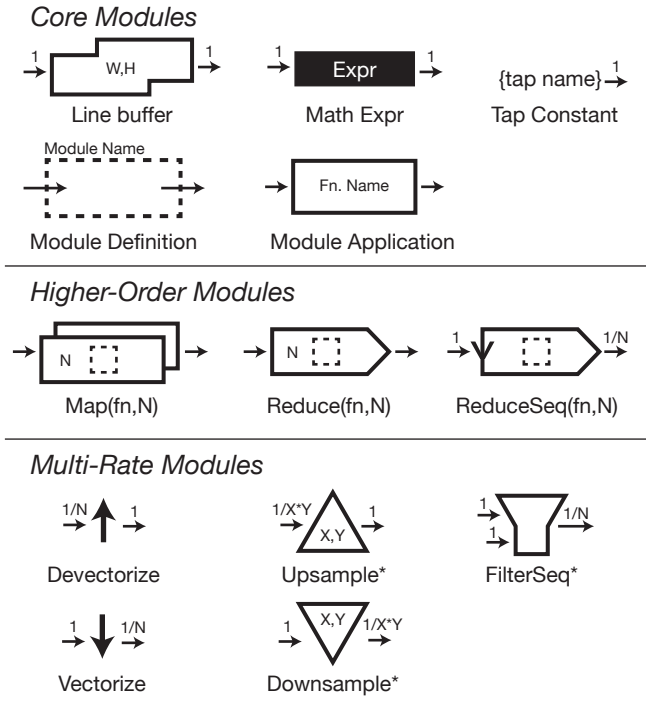


Figure 3: *List of the built-in modules in Rigel. As in SDF, numbers on edges indicate the input and output rates. (*) indicates variable-latency modules.*

Rigel is a higher-level programming model than languages like C. In particular, Rigel performs domain-specific program checking using SDF rules, and contains domain-specific image processing operations such as line buffering. In the future, we may consider HLS as a compile target for Rigel instead of Verilog to simplify our implementation.

3 Multi-Rate Line-Buffered Pipelines

We now describe the multi-rate line-buffered pipeline architecture, and show how it can be used to implement advanced image processing pipelines. Applications are implemented in our system by creating a DAG of instances of a set of built-in static and variable-latency SDF modules. The core modules supported by our architecture are listed in figure 3.

As in synchronous dataflow, each of our modules has an SDF input and output rate. Our modules always have rates $M/N \leq 1$, which indicates that the module consumes/produces a data token on average every M out of N firings. Each data token in our system has an associated type. Our type system supports arbitrary-precision ints, uints, bitfields, and booleans. In addition, we support 2D vectors and tuples, both of which can be nested.

3.1 Core Modules

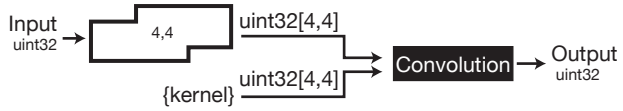
Our architecture inherits a number of core modules from the line-buffered pipeline architecture.

Our **line buffer** module takes a stream of pixels and converts it into stencils. Many of our modules can operate over a range of types. The line buffer has type $A \rightarrow A[stencilW, stencilH]$ for an arbitrary type A, indicating that its input type is A and output type is the vector $A[stencilW, stencilH]$.

A **Math Expr** is an arbitrary expression built out of primitive mathematical operators (+,*,≫, etc). Math exprs also include operations for slicing and creating vectors, tuples, etc. Our math exprs support all of the operators in Darkroom's image functions plus some additional functionality. In particular, we added arbitrary precision fixed-point types to represent non-integer numbers. Since FPGAs do not have general floating point support, we found that better support for fixed-point types was necessary. We also include some primitive floating point support, such as the ability to normalize numbers.

Tap Constants are programmable constant values with arbitrary type. Taps can be reset at the start of a frame, but cannot be modified while a frame is being computed.

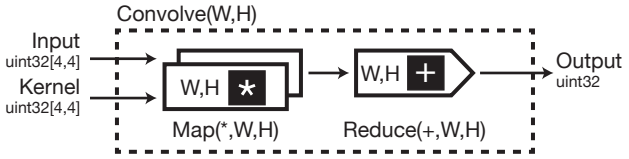
These core modules can be used to implement a pipeline that matches hardware produced by the line-buffered pipeline architecture. For example, we can use a 4×4 stencil line buffer, a math expr that implements convolution (multiplies and a tree sum unrolled), and a tap constant with the convolution kernel to get a line-buffered pipeline:



3.2 Higher-Order Modules

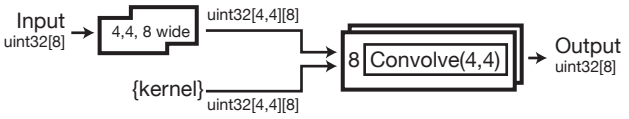
Our architecture has *higher-order modules*, which are modules built out of other modules. **Map** takes a module with type $A \to B$ and lifts it to operate on type $A[N] \to B[N]$ by duplicating the module N times. **Reduce** takes a binary operator with type $\{A, A\} \to A$ and uses it to perform a tree fold over an vector of size N, producing a module with type $A[N] \to A$.

We can use map and reduce to build the convolution function from modules in our architecture, instead of creating it by hand as math ops. We also show a **module definition**, which defines a pipeline so that it can be reused multiple times later. We parameterize the pipeline over stencil width/height. This is not a core feature of our architecture, but is instead accomplished with metaprogramming:



Our higher-order modules can also be used to implement space-time tradeoffs. Implementing space-time tradeoffs in our system involves creating multiple implementations of an algorithm with a range of *parallelisms*. We formally define parallelism, p, to be the width of the datapaths in the pipeline. For example, p=2 indicates that the pipeline can process two pixels per firing, p=1/2 indicates that the pipeline can only do half a pixel's worth of computation per firing.

Here we demonstrate an 8-wide data-parallel implementation of convolution (p=8). We use the map operator to make 8 copies of the *Convolve* module we defined above. The line buffer module shown previously can be configured to consume/produce multiple stencils per firing. To feed this pipeline with data, we configure the runtime system to provide a vector of 8 pixels as input. These changes yield a pipeline that can produce 8 pixels/firing:



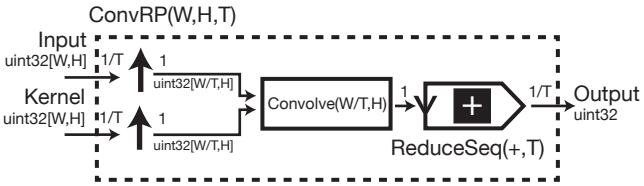
3.3 Multi-Rate Modules

Next we introduce a number of our architecture's multi-rate modules. We first show multi-rate modules that are used to reduce the parallelism of a pipeline (p<1), so designs can trade parallelism for reduced area.

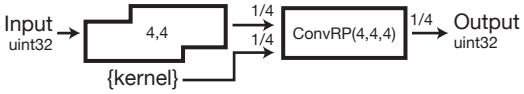
To reduce parallelism, we need to perform a computation on less than a full stencil's worth of data. We accomplish this with the **devectorize** module. Devectorize takes a vector type, splits it into smaller vectors, and then outputs the smaller vectors over multiple firings. With 2D vectors we devectorize the rows. **Vectorize** performs the reverse operation, taking a small vector over multiple firings and concatenating them into a larger vector:



ReduceSeq is a higher-order module that performs a reduction sequentially over T firings (type $A \to A$). Here we combine devectorize (which increases the number of tokens, at lower parallelism) and reduceSeq (which decreases the number of tokens). The convolution module can now operate on stencil size 1×4 instead of 4×4, reducing its amount of hardware by $4\times$. We refer to this pipeline as the reduced parallelism *ConvRP*:



We can connect our new *ConvRP* module to the line buffer and convolution kernel as in the previous examples:



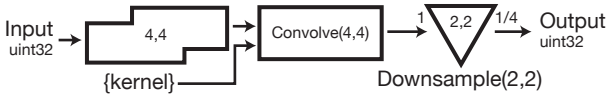
The total throughput of a pipeline is limited by the module instance with the lowest throughput. In this example, *ConvRP* has input/output rate of 1/4, which means that the resulting pipeline can only produce one output every 4 firings.

3.4 Multi-Scale Image Processing Modules

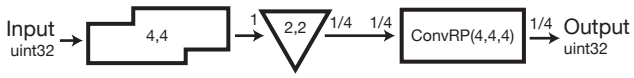
Next, we introduce multi-rate modules in our architecture that are used to implement multi-scale image processing. Rigel's **downsample** module discards pixels based on the user's specified integer horizontal and vertical scale factor (module type $A \to A$). Similarly, Rigel's **upsample** module upsamples a stream by duplicating pixels in X and Y a specified number of times (module type $A \to A$):



We can use these modules to downsample following a convolution, which is a component of a pipeline for computing Gaussian pyramids. A basic implementation simply adds a downsample module to the convolution example shown previously:



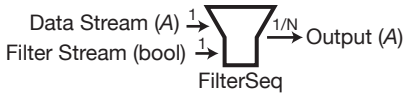
This implementation is suboptimal, however, because the convolution hardware is sized to produce 1 output pixel/firing, but is only used for 1/4 the pixels. When we perform the SDF solve on this pipeline, we will see that the firing rate for *Convolve* is 1/4, indicating it will sit idle 3/4 of the cycles. We can use the reduced parallelism *ConvRP* module from the previous example to increase the efficiency of the design:



Because *ConvRP(4,4,4)* has input/output rate 1/4, it matches the rate of the downsample, and all hardware does useful work every cycle.

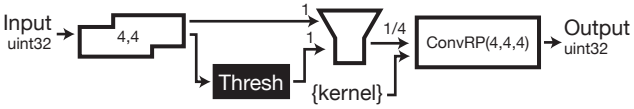
3.5 FilterSeq Module

Next we introduce a module that can be used to implement sparse computations. **FilterSeq** takes two inputs: a stream of data of type A to be filtered and a stream of booleans. FilterSeq only outputs data when the boolean stream is true:



To fit into the synchronous dataflow model, filterSeq overrides the user's boolean function if it produces too few or too many total outputs. In addition, to fit into the bounded latency restriction of the variable-latency SDF model, filterSeq overrides the user's boolean function if it has not produced an output within a certain number of firings. Users can tune the amount of latency-variance each filterSeq instance allows to suit their application, with more variance requiring a larger FIFO.

The filterSeq module can be used to implement a sparse convolution. The *Thresh* function outputs boolean true if the center pixel in the stencil is above a certain threshold (type $uint32[4,4] \rightarrow bool$). As in the previous example, we use the reduced parallelism *ConvRP* module for efficiency, because the stream after the filterSeq runs at 1/4 the rate of the input. The resulting pipeline reduces the input data stream by 1/4:



3.6 Additional Modules

Rigel also includes a number of additional modules. **pad** and **crop** are used to pad and crop 2D vectors. We use this to implement image boundary conditions. When combined, pad and crop reduce the throughput of the entire pipeline slightly, which corresponds to the extra cycles required to prime the pipeline with boundary values. **Serialize** takes multiple pixels streams and serializes them into a single stream based on a user-specified ordering function. We use this to write out an image pyramid into a human-readable format. Rigel also includes a number of fused modules (e.g., a fused line buffer and devectorize) which implement the same functionality at reduced hardware cost.

4 Scheduling Model

Rigel's compiler takes an application specified in the multi-rate line-buffered pipeline architecture and lowers it to a concrete FPGA implementation. Our compiler allows for manual control of a number of aspects of the hardware that we generate, so that the user can tweak their implementation to get the best performance. In particular, we allow for manual specification of static vs dynamic scheduling, and FIFO sizes.

4.1 Static vs Dynamic Scheduling

Rigel supports generating hardware for both statically-scheduled pipelines and dynamically-scheduled pipelines. Statically-scheduled pipelines behave like the pipelines in Darkroom, where each module assumes that its inputs have valid data every cycle, and that downstream modules are ready to read every cycle [Hegarty et al. 2014]. In contrast, dynamically-scheduled pipelines have additional hardware to check the validity of inputs, and stall if downstream modules are not ready.

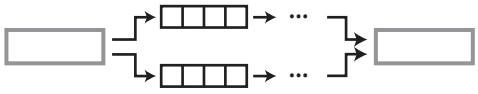
For every module instance, the user specifies whether they want it to be statically or dynamically scheduled. Mult-rate modules, such as downsample and upsample, can only be dynamically scheduled.

4.2 FIFO Allocation

Rigel gives the user manual control over FIFO placement and sizing. Manual control allows the user to to make performance tradeoffs, such as reducing FIFO size when it has a small effect on runtime.

FIFOs only need to be manually allocated for dynamically-scheduled pipelines. Scheduling for statically-timed pipelines is performed automatically by our compiler using standard retiming techniques [Leiserson and Saxe 1991; Hegarty et al. 2014]. FIFOs must be added to a dynamically-scheduled pipeline for two reasons:

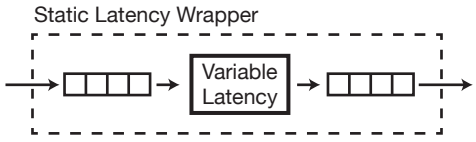
Delay Matching



As in statically-timed hardware pipelines, DAG fan-out and reconvergence requires delay matching along the paths so that data can arrive at the reconvergence point at the same time. Because FIFOs adjust their size dynamically, the user does not need to match delays exactly, only provide an upper bound.

Rigel has two sources of delay along paths that must be accounted for: hardware pipeline delay, and data decimation (e.g., modules like crop). We have found that adding a minimum size (128 element) FIFO along each branch is sufficient for most cases. Hardware pipeline delay is usually much less than 128 cycles, and pipelines that do data decimation differently along branches are rare.

Variable Latency Hiding



As explained in section 2.2, we allow SDF modules in our architecture to have variable, but bounded, latency. Downsample in Y for example would produce pixels for one whole line, and then sit idle for one whole line. This can cause performance problems with neighboring modules. For example, a downstream module that expects an input every $1/2$ cycle would sit idle more than expected,

because it would have no data to work on while the downsampleY produces an empty line.

To fix these performance problems, we wrap these variable-latency modules with a FIFO that is large enough to absorb the variability. Because the variability is bounded, this is always possible by definition. To assist in sizing these FIFOs, we implemented a simple simulator for a number of our variable-latency modules to measure the FIFO size needed to hide their variability.

FIFOs can also serve to improve clock cycle time. All of our dynamically-scheduled modules are driven by a stall signal from downstream. If many modules are driven by the same stall, this can limit the clock period of the design. To solve this, the user can insert a FIFO to break the stall into different domains. In addition, if the FIFO is sized large enough that it will never fill, we allow the user to disable the stall signal entirely.

5 Implementation

We implemented a compiler that takes a multi-rate line-buffered pipeline (sec. 3) along with scheduling choices (sec. 4) and lowers it to Verilog. Our multi-rate line-buffered pipeline construction library and compiler is embedded in Lua. An embedded implementation is convenient because it allows for easy metaprogramming of the pipelines, which we use to create parameterized pipelines.

5.1 Semantic Checking

Prior to lowering to Verilog, we run a simple typechecker that checks that types along edges in the pipeline match and that dynamic vs static scheduling options match.

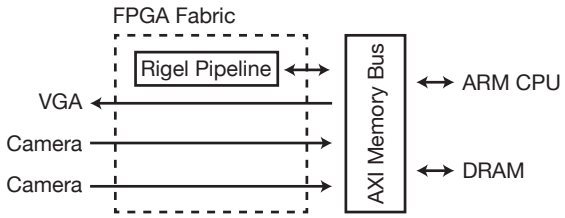
We also run standard SDF scheduling [Lee and Messerschmitt 1987]. SDF performs two important functions in our compiler. First, we use it to check for malformed pipelines that will deadlock, as explained in section 2.2. Second, the SDF firing rates for each module correspond to the throughput the module will have at runtime. The user can use these rates to determine if they have any underutilized modules, and improve the efficiency of their design by applying the techniques shown in section 3. This saves the user from having to execute an expensive simulation or synthesis process to determine pipeline performance characteristics.

5.2 Lowering to Verilog and Simulation

Because all of our built-in modules match the desired hardware closely, lowering to Verilog is straightforward. For each of the modules shown in section 3, we implement a function to lower them to a Register Transfer Level (RTL) intermediate of our own design. RTL is a low-level language that specifies hardware as registers and the circuits that drive them. For statically-scheduled pipelines, we then perform a pipelining transform on the RTL to improve clock cycle time, using standard techniques [Leiserson and Saxe 1991]. Finally, we translate our RTL representation to Verilog.

We also use the Terra language to lower our graph of modules to a near-cycle-accurate CPU simulation [DeVito et al. 2013]. Each hardware module yields a Terra class that implements it, with a cycle-accurate clock-tick simulation function. Our simulator is designed for accuracy, not speed, but it is many times faster than Verilog simulation.

5.3 Camera Test Rig



We implemented a camera test rig on the Xilinx Zynq System On a Chip (SOC) platform to test and evaluate our system (above). All devices communicate through a standard ARM AXI memory bus. Two Omnivision OV7660 cameras and VGA are driven by lightweight controllers implemented in the FPGA fabric. The ARM core runs standard Linux, and is used to configure devices on the FPGA fabric using memory-mapped IO (e.g., framebuffer sizes/locations, camera registers, start/stop instructions).

We synthesized bitstreams from our Verilog for the Zynq 7020 and 7100 using Xilinx ISE 14.5. The Zynq 7020 is an entry-level FPGA costing around \$300 ($\sim$1.3M ASIC gate equivalent), and the Zynq 7100 is a high-end FPGA costing around \$2000 ($\sim$6.6M ASIC gate equivalent) [Xil 2016]. We disabled DSP slices (fixed ALUs) for our synthesis to make slice numbers across different configurations more comparable. We configured the clocking infrastructure of each board to execute our tests at the numbers we report.

We implemented two system configurations on this SOC platform. First, we have a test configuration that writes the input image into a DRAM framebuffer using Linux, executes the Rigel pipeline on the framebuffer, and reads the output framebuffer in Linux to save to disk. We automatically run 100s of directed and integration tests using this setup to verify correctness of our compiler and modules. We check that the output of each test executed in the Terra simulator, Verilog simulator, and on the board match exactly. Second, we have a live demo configuration that has 1 or 2 camera(s), VGA, and the Rigel pipeline configured to write into framebuffers in a continuous streaming fashion. We demonstrate our camera/display configuration of the test rig running a camera pipeline and depth from stereo in the accompanying video.

6 Evaluation

We implemented a number of image processing algorithms in our system, and used Rigel to synthesize hardware designs for them running on the Zynq 7020 and Zynq 7100 FPGAs.

CONVOLUTION is an implementation of a single 8×8 convolution. This example will be used to demonstrate the scalability of our system, from very small to very large pixel throughputs.

STEREO is a simple implementation of depth from stereo based on brute force search [Scharstein and Szeliski 2002]. This example operates on a stereo pair of images. We search 64 neighboring pixels in the second camera to find the offset with the lowest 8×8 Sum of Absolute Difference (SAD). Our brute force search is extremely regular and compute intensive, and thus demonstrates the advantage of FPGAs compared to CPUs/GPUs on this type of workload.

FLOW is an implementation of the Lucas-Kanade optical flow algorithm [Lucas et al. 1981]. We use 1 iteration of Lucas-Kanade to compute dense flow information with a 12×12 window search. Lucas-Kanade is challenging on FPGAs because it performs a floating point matrix inversion. For this example, we used Rigel's fixed-point types to create an equivilant fixed-point implementation. The

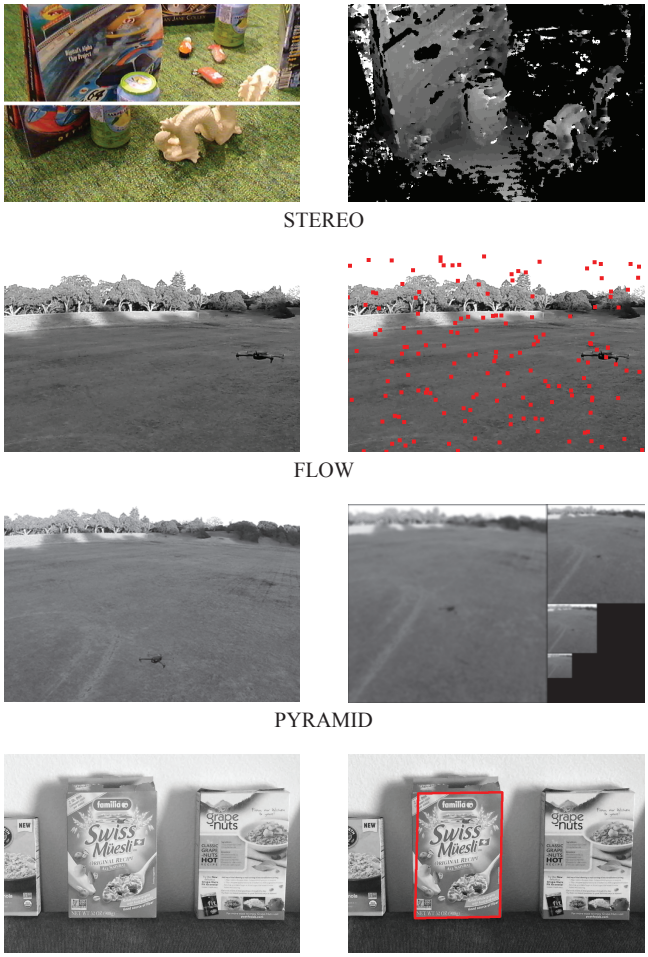


Figure 4: *We implemented a number of fundamental image processing algorithms in Rigel.* STEREO *and* FLOW *test Rigel's chip area scaling.* PYRAMID *and* DESCRIPTOR *test Rigel's multi-rate modules.*

output of our integer implementation only differs from the floating point version by a few bits.

PYRAMID computes a pyramid of 8×8 Gaussian blurs. Pyramids are a core component of many image processing algorithms [Adelson et al. 1984]. We use this example to evaluate the effectiveness of our space-time tradeoff techniques and dynamic scheduling.

DESCRIPTOR is an implementation of the feature descriptor in the Scale Invariant Feature Transform (SIFT) algorithm [Lowe 1999]. We use Rigel's filterSeq module to compute the descriptors sparsely at Harris corners [Harris and Stephens 1988]. This example is used to evaluate how sparse computations perform in our system.

6.1 Space-Time Tradeoffs

To evaluate the effectiveness of Rigel at supporting space-time tradeoffs, we implemented CONVOLUTION, STEREO, and FLOW at a range of differents parallelisms (p), as defined in section 3.2. Ideally, chip area should scale linearly with parallelism, and execution time should scale inversely with parallelism. For each example program, we will use the p=1 case as a reference point, and expect $Area_p = Area_1 * p$ and $Cycles_p = Cycles_1/p$.

We show the area scaling of CONVOLUTION, STEREO, and FLOW normalized to their size at p=1 in figure 5. The p>1 cases are

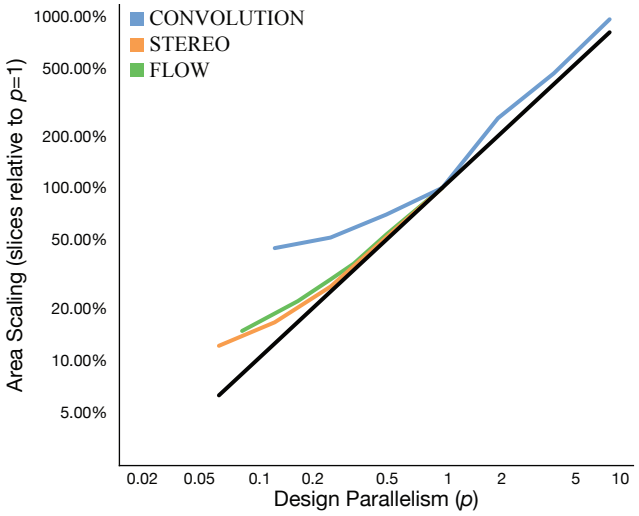


Figure 5: *Area scaling on the Zynq 7100, shown as slices normalized to the slices at parallelsim 1. Ideally, area should scale linearly with parallelism. Designs deviate from this ideal scaling at low parallelism, where non-scalable hardware like the line buffer starts to dominate the cost.*

implemented using the technique in section 3.2, and the p<1 cases are implemented as in section 3.3. STEREO and FLOW are only displayed for $p \leq 1$ because this is the largest size that fits on the 7100. We see that area is close to expected at high parallelisms, but that area is higher than expected at low parallelisms. Area scaling efficiency is limited by the percentage of hardware that can actually be scaled, which decreases at low parallelisms. This effect is particularly noticeable with CONVOLUTION: at low parallelism the total amount of hardware generated is so small (1.06% of the slices) that small non-scalable hardware like the line buffer starts to dominate the cost.

CONVOLUTION (1080p)

p	SDF Px/Cyc	Measured Px/Cyc	% Inc	FIFO KBs
1/8	0.1236	0.1236	0%	0
1	0.9885	0.9885	0%	2.00
4	3.9541	3.9539	0%	8.00

STEREO (720x400)

p	SDF Px/Cyc	Measured Px/Cyc	% Inc	FIFO KBs
1/16	0.0550	0.0550	0%	0.625
1/4	0.2200	0.2200	0%	0.625
1	0.8801	0.8800	0.01%	2.25

FLOW (1080p)

p	SDF Px/Cyc	Measured Px/Cyc	% Inc	FIFO KBs
1/12	0.0818	0.0818	0%	20.6
1/6	0.1637	0.1637	0%	36.7
1	0.9820	0.9819	0%	4.00

Figure 6: *Predicted and measured pixels/cycle for each design over a range of parallelisms. We see that predicted and measured pixels/cycle match the parallelism (p) closely. On these examples, FIFOs are only necessary to match delays and improve clock timing, which results in small FIFO sizes.*

Next, we evaluate execution time scaling of CONVOLUTION, STEREO, and FLOW. Ideally, the average pixels/clock of the pipeline should be equal to p. Figure 6 shows the pixels/clock predicted by the SDF model. The SDF prediction number is the

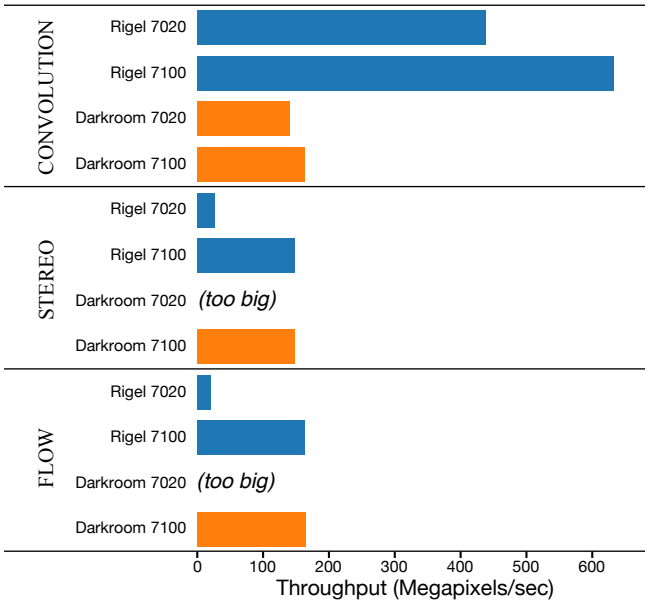


Figure 7: *Each of the example programs synthesized to the highest throughput that fits on each FPGA board. Compared to Darkroom, Rigel is able to scale up the parallelism of memory-bound computations like* CONVOLUTION *to get higher throughput. Rigel is also able to scale down compute-bound examples like* STEREO *and* FLOW *to fit on the smaller Zynq 7020.*

throughput of the lowest throughput module found in the SDF solve, which limits the throughput of the entire pipeline. We see that the pixels/clock predicted by the SDF model shown in figure 6 are very near p. Small deviations from p are due to extra cycles needed to compute the boundary region (sect. 3.6).

We also report the pixels/cycle attained by running the design on the FPGA board (including DRAM stalls, etc). We see that the measured pixels/cycle rate is almost identical to the value predicted by SDF (fig. 6). This indicates that our dynamic scheduling hardware is working correctly. We also see that the FIFO sizes needed to attain these measured pixels/clock rates are low for these examples. In these designs, FIFOs were only inserted to improve clock cycle timing, or to match delays along branches.

Finally, we report the highest throughput ($clock{*}pixels/cycle$) implementation for each board synthesized and run using our system (fig. 7). We see that CONVOLUTION can support 632 megapixels/second on the 7100, STEREO can support 148 megapixels/second and FLOW can support 165 megapixels/second.

We also compare our results to Darkroom, which always synthesizes pipelines with parallelism p=1. We see that for small memory-bound computations like CONVOLUTION, Darkroom cannot take advantage of the extra compute available on the FPGA. For large designs like STEREO and FLOW, Darkroom cannot synthesize designs for the 7020 because it has no way to reduce the area of these examples to fit on this smaller board.

In figure 8, we break down p, the clock rate, and resource utilization of the highest-throughput designs measured in figure 7. We see that Rigel generates designs that achieve a typical clock rate for FPGAs, indicating that our automatic pipelining works correctly, and that our generated Verilog has no problems that limit the clock rate. Finally, we see that the % utilization of the FPGA slices is consistently high (too large to double), with the exception of CONVOLUTION, which is limited by the width of the memory bus. This indicates that Rigel's flexible programming model allows us to use the compute resources available.

Zynq 7020

Pipeline	p	Clock	Slices	BRAMs
CONVOLUTION	4	111Mhz	39%	6.4%
STEREO	1/4	125Mhz	96%	5.0%
FLOW	1/6	125Mhz	79%	25.4%

Zynq 7100

Pipeline	p	Clock	Slices	BRAMs
CONVOLUTION	4	160Mhz	7%	1.2%
STEREO	1	169Mhz	65%	1.0%
FLOW	1	169Mhz	50%	4.7%

Figure 8: *Rigel's synthesized designs have typical clock periods for large designs on FPGAs. For compute-bound applications like* STEREO *and* FLOW*, Rigel successfully synthesizes designs that use the majority of the compute resources available on the FPGA. BRAMs are used to implement line buffers and FIFOs, and are not a limiting factor for these designs.*

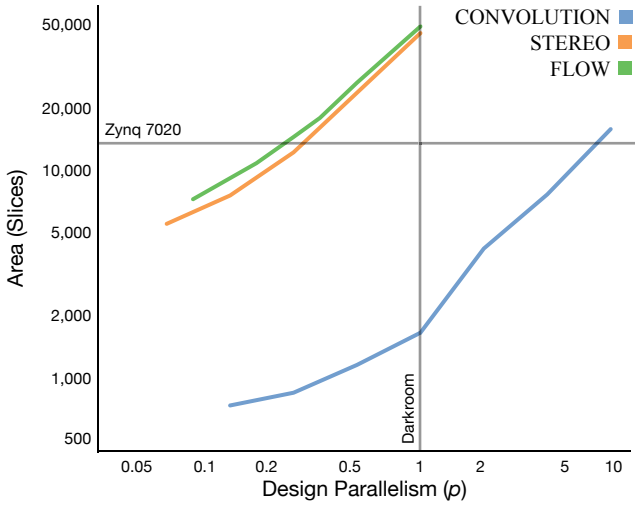


Figure 9: *Summary of the entire design space supported by Rigel. Points above the Zynq 7020 line use too many slices to fit on the smaller Zynq. Darkroom only supports a single parallelism in this design space, which couples chip area to algorithm complexity.*

Finally, we summarize the entire design space supported by Rigel in figure 9, showing different algorithms, parallelisms, and their resulting areas. We see that Rigel's space-time tradeoffs allows us to decouple chip area from algorithm complexity.

6.2 Performance Comparison

While not intended to be a full evaluation of the merits of FPGAs compared to CPUs/GPUs, we want to estimate our system's performance relative to some existing low-power platforms that are available today. We evaluate Rigel on FPGA against the four core 2.32 Ghz ARM Cortex A15 on the Nvidia Jetson TK1.

In figure 10 we see that Rigel on the Zynq 7020 is between 3×-55× faster than the A15 on our two computer vision applications, STEREO and FLOW. Crucially, this improved performance allows the pipelines to run on real image sizes, such as 720p/30fps and 640×480/60fps, whereas the ARM can only support at most

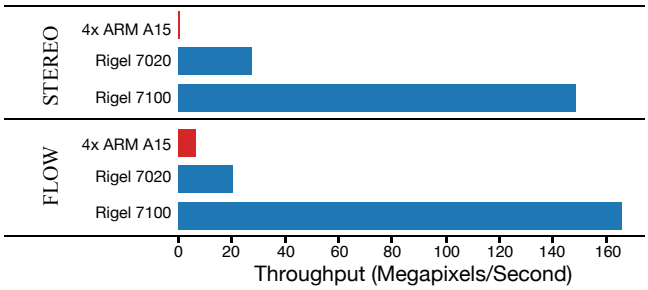


Figure 10: *We compared Rigel on FPGA to multi-threaded and vectorized CPU implementations of our applications compiled using Darkroom running on a 4 core Cortex A15 . The 7020 is 3×-55× faster than the A15. The 7100 is 25×-297× faster than the A15.*

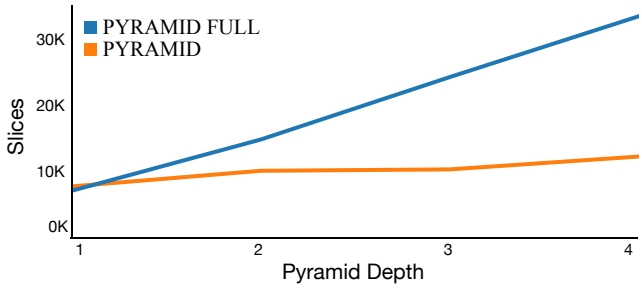


Figure 11: PYRAMID FULL *and* PYRAMID *over a range of pyramid depths.* PYRAMID *can implement a deep pyramid using only 1.58× the slices, compared to 4.67× with* PYRAMID FULL*, but at the expense of some reduction in throughput.*

PYRAMID FULL (384x384, P=4)

Depth	SDF Px/Cyc	Measured Px/Cyc	% Inc	FIFO KBs
1	3.8384	3.8358	0.07%	16
2	3.8384	3.8035	0.92%	296
3	3.8384	3.7640	1.98%	576
4	3.8384	3.6930	3.94%	856

PYRAMID (384x384, P=4)

Depth	SDF Px/Cyc	Measured Px/Cyc	% Inc	FIFO KBs
1	3.8384	3.8358	0.07%	16
2	3.8384	3.7940	1.17%	296
3	3.6864	3.0620	20.39%	346
4	3.6864	2.9591	24.58%	396

Figure 12: *Predicted and measured pixels/cycle of pyramid pipelines on the Zynq 7100. These pipelines amplify data, so Pixels/cycle reports the number of input pixels read and processed per cycle. There is a serial dependency between pyramid levels, which causes measured pixels/cycle to diverge from predicted on deeper pyramids. FIFOs are used to hold stencil intermediates before they are processed or written out, which results in large FIFO size.*

640×480/20fps on FLOW. The FPGAs show more of a speedup on STEREO than FLOW because they can perform specialized low-precision integer ops on STEREO, but must emulate floating point math with expensive high-precision integers on FLOW.

We also compared STEREO to the brute force depth from stereo example in the CUDA SDK running on the Jetson TK1's GPU. CUDA Stereo is highly optimized, making use of shared memory and a 4 channel SAD intrinsic which performs 7 math ops in 1 cycle. On a 4 channel 8×8 configuration, the TK1 performs stereo at 4.2 Megapixels/second compared to 3.16 Megapixels for the 7020. However, on a 1 channel 8×8 configuration the TK1 no longer gets an advantage from the SAD intrinsic, so it performs at 4.2 Megapixels/seconds, compared to 27 Megapixels/second on the 7020. The TK1 uses 5-10 watts total board power depending on load [Elinux 2015], compared to ∼5 watts total board power for the Zynq 7020. On the high end, the Nvidia Titan Black (∼225 watts) performs 1 channel 8×8 STEREO at 128 Megapixels/second, compared to the same configuration on the Zynq 7100 (30 watts or less, [Xilinx 2016]) running at 148 Megapixels/second. These results indicate that FPGAs can have a significant performance/watt advantage on some applications compared to a GPU.

6.3 Pyramid Image Processing

To evaluate Rigel's ability to support pyramid workloads, we implemented two variants of a Gaussian pyramid, PYRAMID FULL and PYRAMID. In PYRAMID FULL, each pyramid depth is computed in parallel with the same parallelism factor. So, we expect the chip area to grow linearly for each pyramid depth. This is highly inefficient however, as the deeper levels of the pyramid process much less data, so the hardware will be sitting idle for many cycles. To solve this problem, PYRAMID applies the parallelism reduction techniques from section 3.3. The pipeline for one level of this example is presented in section 3.4.

To compare PYRAMID FULL and PYRAMID, we synthesized each at a range of depths. Figure 11 shows the number of slices for each depth. As expected, PYRAMID FULL's chip area grows by a large amount for each depth (a 4.67× increase from depth 1 to 4). In contrast, PYRAMID grows less (a 1.58× increase from depth 1 to 4).

Next, we report the predicted and measured pixels/cycle for PYRAMID FULL and PYRAMID in figure 12. Because PYRAMID FULL is computed in parallel with excess parallelism, we see that it is predicted to generate the pyramid in the same amount of time regardless of pyramid depth. As in the pipelines in section 6.1, predicted pixels/cycle is slightly less than the parallelism ($p = 4$) due to the boundary region calculation. On smaller images, the boundary region is a higher percentage of the runtime. This effect, combined with parallelism reduction causes deeper pyramid versions of PYRAMID with smaller images to have slightly lower predicted pixels/cycle.

Unlike the examples in section 6.1, PYRAMID FULL and PYRAMID have composed stencil operations, which cause there to be serial dependencies between layers of the pyramid. Deeper pyramid levels cannot proceed until a stencil's worth of pixels in the finer level have been computed. SDF does not model latency effects like these. The SDF prediction is calculated with all modules in the pipeline proceeding immediately in parallel. Because of this, we see that both PYRAMID and PYRAMID FULL have higher measured pixel/cycle than predicted. This effect is particularly pronounced with PYRAMID, whose deepest level is running at much lower throughput, so takes longer to complete given a late start time.

Finally, we show the total throughput of PYRAMID FULL and PYRAMID in figure 13. Both implementations have similar performance, with minor differences due to clock cycle time and different pixels/cycle. In addition, we show the resource utilization of our pyramid implementations in figure 14. Most significantly, we see that the optimizations in PYRAMID allow it to fit on the Zynq 7020, whereas PYRAMID FULL takes up a large percentage of the area of the 7100.

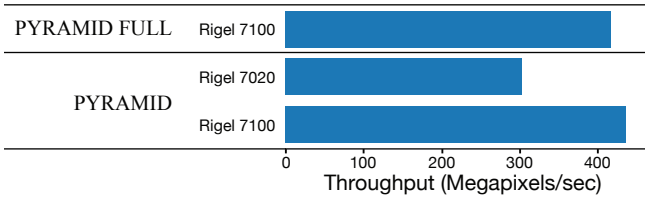


Figure 13: *Measured throughput of* PYRAMID *and* PYRAMID FULL. *All three implementations have the same parallelism, so differences in performance are due to differences in clock and execution cycles.*

p	Clock	% Slices	% BRAMs
PYRAMID FULL (Zynq 7100)			
4	85 Mhz	49%	32.1%
PYRAMID (Zynq 7020)			
4	77 Mhz	85%	85.7%
PYRAMID (Zynq 7100)			
4	111 Mhz	16%	22.3%

Figure 14: *FPGA resource utilization for pyramid pipelines.* PYRAMID *reduces the number of slices sufficiently compared to* PYRAMID FULL *that the design is able to fit on the Zynq 7020.*

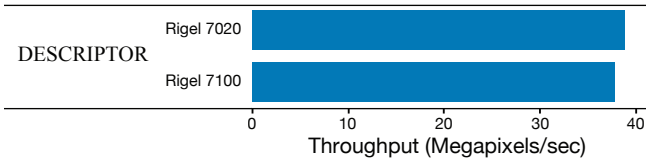


Figure 15: *Measured throughput of* DESCRIPTOR. *Both boards are running the same design, so they have similar performance, 38 Megapixels/sec.*

DESCRIPTOR (1080p)

Board	p	SDF Px/Cyc	Measured Px/Cyc	FIFO KBs
7020	1/2	0.50000	0.49646	283
7100	1/2	0.50000	0.49646	283

Board	Clock	% Slices	% DSP48s	% BRAMs
7020	77 Mhz	86%	58%	70%
7100	75 Mhz	17%	6%	13%

Figure 16: *Runtime and synthesis statistics for* DESCRIPTOR. *Rigel's synthesized hardware is able to execute this sparse, variable-latency pipeline in very near to the predicted number of cycles. DSP48s were used to emulate floating point operations, which are not natively supported by the Zynq FPGA.*

6.4 Sparse Computations

To demonstrate how Rigel performs on sparse computations, we implemented DESCRIPTOR, a sparsely-computed feature descriptor based on the feature descriptor in SIFT. We report the predicted and measured pixels/cycle for DESCRIPTOR in figure 16. DESCRIPTOR takes 256 cycles to calculate each feature descriptor, and calculates descriptors sparsely on average once every 128 pixels, yielding $p{=}1/2$. We see that both the SDF prediction and measured pixels/cycle closely match the expected number of cycles. FIFOs are used to hide the variable latency of the filterSeq operator, and do not use a prohibitive amount of buffering (283KBs). We show in the accompanying video that implementing Harris corners within the restrictions of our static-rate bounded-latency filterSeq module (sec. 3.5) does not prevent this pipeline from reliably tracking single-scale objects.

DESCRIPTOR performs roughly 60 total floating point operations. The Zynq FPGA platform does not have native support for floating point operations, so we emulated these operations using a Xilinx floating-point library. This results in relatively high hardware usage per op compared to integer pipelines (fig. 16). Despite this cost, both boards are able to execute this pipeline at 38 Megapixels/second (fig. 15). We believe FPGA vendors will add better support for floating point operations in the near future.

7 Discussion and Future Work

Rigel takes pipelines specified in the multi-rate line-buffered pipeline architecture and compiles them to FPGA designs. Efficient hardware implementation of advanced image processing and vision algorithms necessitates a system that can support image pyramids, sparse computations, and space-time implementation tradeoffs. We showed how the simple multi-rate primitives in our architecture can support all three features simultaneously. Our implementations of depth from stereo, Lucas-Kande, Gaussian pyramids, and the SIFT descriptor use these techniques to achieve high performance on FPGA platforms. Our system is able to support these applications end-to-end, from a high-level pipeline description to an FPGA design running at 20–463 megapixels/second.

Implementing image processing applications in Rigel is faster and higher-level than in existing hardware languages like Verilog. In the future, we would like to examine ways to make Rigel's compiler infrastructure even more productive and convenient. In particular, while our compiler uses SDF to inform the user what modules in their pipeline are running at low throughputs, it requires them to fix these problems manually or with metaprogramming. We also require the user to manually specify scheduling choices like FIFO sizes. We think that in both these cases some simple heuristics could be used to save programmer effort while still providing sufficient performance.

While we demonstrated the viability of our system with a camera test rig, in the future we would like to see this developed into a full, user-friendly research platform, possibly as an extension to existing models for programmable cameras [Adams et al. 2010]. We envision this platform allowing the user to easily configure and run different multi-camera and output setups, making hardware and FPGA design for image processing accessible to a wider audience.

8 Acknowledgments

Thanks to Niels Joubert for help with the camera test rig and quadcopter video. This work has been supported by the DOE Office of Science ASCR in the ExMatEx and ExaCT Exascale Co-Design Centers, program manager Karen Pao; DARPA Contract No. HR0011-11-C-0007; DARPA Agreement No. FA8750-14-2-0009; fellowships and grants from NVIDIA, Intel, and Google; and the Stanford Pervasive Parallelism Lab (supported by Oracle, AMD, Intel, and NVIDIA). Any opinions, findings and conclusion or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

ADAMS, A., TALVALA, E.-V., PARK, S. H., JACOBS, D. E., AJDIN, B., GELFAND, N., DOLSON, J., VAQUERO, D., BAEK,

J., TICO, M., LENSCH, H. P. A., MATUSIK, W., PULLI, K., HOROWITZ, M., AND LEVOY, M. 2010. The Frankencamera: An experimental platform for computational photography. *ACM Transactions on Graphics 29*, 4 (July), 29:1–29:12.

ADELSON, E. H., ANDERSON, C. H., BERGEN, J. R., BURT, P. J., AND OGDEN, J. M. 1984. Pyramid methods in image processing. *RCA engineer 29*, 6, 33–41.

BILSEN, G., ENGELS, M., LAUWEREINS, R., AND PEPERSTRAETE, J. 1995. Cyclo-static data flow. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, 3255–3258.

BOUGUET, J.-Y. 2001. Pyramidal implementation of the affine Lucas Kanade feature tracker description of the algorithm. Tech. rep., Intel Corporation.

BRUNHAVER, J. 2015. *Design and Optimization of a Stencil Engine*. PhD thesis, Stanford University.

DEVITO, Z., HEGARTY, J., AIKEN, A., HANRAHAN, P., AND VITEK, J. 2013. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 105–116.

ELINUX, 2015. Jetson computer vision performance. http://elinux.org/Jetson/Computer_Vision_Performance. [Online; accessed 12-April-2016].

HAMEED, R., QADEER, W., WACHS, M., AZIZI, O., SOLOMATNIKOV, A., LEE, B. C., RICHARDSON, S., KOZYRAKIS, C., AND HOROWITZ, M. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ACM, 37–47.

HARRIS, C., AND STEPHENS, M. 1988. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, 147–151.

HEGARTY, J., BRUNHAVER, J., DEVITO, Z., RAGAN-KELLEY, J., COHEN, N., BELL, S., VASILYEV, A., HOROWITZ, M., AND HANRAHAN, P. 2014. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics 33*, 4 (July), 144:1–144:11.

HORSTMANNSHOFF, J., GROTKER, T., AND MEYR, H. 1997. Mapping multirate dataflow to complex rt level hardware models. In *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*, 283–292.

HUANG, J., QIAN, F., GERBER, A., MAO, Z. M., SEN, S., AND SPATSCHECK, O. 2012. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, ACM, 225–238.

LEE, E. A., AND MESSERSCHMITT, D. G. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers 100*, 1, 24–35.

LEISERSON, C. E., AND SAXE, J. B. 1991. Retiming synchronous circuitry. *Algorithmica 6*, 1-6, 5–35.

LOWE, D. 1999. Object recognition from local scale-invariant features. In *The Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, 1150–1157 vol.2.

LUCAS, B. D., KANADE, T., ET AL. 1981. An iterative image registration technique with an application to stereo vision. In *International Joint Conference on Artificial Intelligence*, vol. 81, 674–679.

MULLAPUDI, R. T., ADAMS, A., SHARLET, D., RAGAN-KELLEY, J., AND FATAHALIAN, K. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics 35*, 4 (July).

MURTHY, P. K., AND LEE, E. 2002. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing 50*, 8, 2064–2079.

MURTHY, P., BHATTACHARYYA, S., AND LEE, E. 1997. Joint minimization of code and data for synchronous dataflow programs. *Formal Methods in System Design 11*, 1, 41–70.

RAGAN-KELLEY, J., ADAMS, A., PARIS, S., LEVOY, M., AMARASINGHE, S., AND DURAND, F. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics 31*, 4, 32.

SCHARSTEIN, D., AND SZELISKI, R. 2002. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision 47*, 1-3, 7–42.

SUGERMAN, J., FATAHALIAN, K., BOULOS, S., AKELEY, K., AND HANRAHAN, P. 2009. Gramps: A programming model for graphics pipelines. *ACM Transactions on Graphics 28*, 1 (Feb.), 4:1–4:11.

VIVADO, 2016. Vivado high-level synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/. [Online; accessed 12-April-2016].

XILINX. 2016. *Zynq-7000 All Programmable SoC Overview*. DS190 Rev. 1.9.

XILINX, 2016. Power efficiency. http://www.xilinx.com/products/technology/power.html. [Online; accessed 12-April-2016].